1 Introduction to the syntax of Python. Part I

Welcome to Math 488/688: Numerical Analysis.

As the title of the course suggests this semester we will be dealing with *numerical* aspects of solving various mathematical problems. At a first approximation these aspects may be described as reformulating the mathematical problem at hand in a form such that a significant number of simple and yet frequently tedious computations are required. After it one usually performs these computations by hand, or with a calculator, or on a computer, using some programming language. It is the last option that I picked for this course, since we are really limited in the amount of computations we can perform by hand or with a calculator. My intention, however, is to show you how things work *in reality*, and since it is universally done on a computer there will be a significant programming part of the course.

There is no prerequisite for you to be proficient in programming in Python (or any other programming language), we will discuss all the necessary details. This is why I start this course not with mathematics, but with some basic ideas how we can realize simple *algorithms* (an *algorithm* is a finite sequence of well-defined, computer-implementable instructions, typically to solve a class of specific problems or to perform a computation) using Python programming language. I will split this introduction into two parts not to make the first section excessively long.

First step is be able to run Python programs on your computer. Here is one possible way of doing this (you can certainly choose a different one, however, your output, which will be graded, should be presented in the form of error-free executable programs written in Python 3.*). On the web page `https://www.anaconda.com/products/individual` you can find the link to Anaconda Individual Edition, please make sure that you choose correct version (Windows, Mac, or Linux), follow the link, download the program and install it. After it you should be able to access a variety of different tools, the one I recommend is the *Jupyter notebook*, which you can start through Anaconda Navigator, Anaconda prompt, or, even better, by copying its icon on your desk and clicking on it. This will run a virtual server on your computer though your default Internet browser.

First, Jupyter dashboard will be open, where you can see your previous work, download new files, etc. If it is done for the first time, find the button `New`, choose `Python 3`, and this will open a new tab with a new empty notebook, where you can run your code, and also type some text and formulas (actually, there are books written using Jupyter notebook). For example, just type in the cell `2 + 3` and press Shift+Enter. You should be able to see the result of your computation. Try something else, like `2**5`. What does command `**` do? All of this should be pretty straightforward, and I encourage you to find some tutorial how to work with Jupyter notebook online, there are plenty of them, and experiment a lot.

So, since now we know how in principle we can request our computer to do some computations, time to start learning a language to do it, to translate our human (for our course, mathematical) statements into something, which can be interpreted by the computer to produce the desired result. The programming language I picked is Python, because it is powerful, general, relatively easy to learn, and free (it is also quite fashionable nowadays, especially in data science and machine learning).

Math 488/688: Numerical Analysis
e-mail: artem.novozhilov@ndsu.edu. Fall 2021

1.1 A first program in Python

For at least 40 years the very first program that illustrates the basic syntax of a programming language is the one that types the sentence "Hello, world" on the screen. Here is the text of such program in Python:

```
#Program that prints "Hello, World" on the screen

print("Hello, World")
```

Let me parse it. First, the first line (green) is not necessary, it is an example of a commentary in Python, it starts with `#` sign. It is usually a good practice to use commentaries while writing your code, it will help you (or someone else who is reading your code) to recall what exactly is done at this particular spot. Next we have a built-in function `print` that, naturally, prints the result of its application on the screen. Note that I use usual parenthesis to accept the input, and the input is simply the string `"Hello, World"`. The fact that we have quotes around our sentence emphasizes that we are dealing with a string, which is a way to represent a lot of different things.

Let's do something a little more complicated. Sometimes we need to provide our program with some data, which we would like to type from keyboard. For this purpose I can use built-it function `input()` (note, I always need to use parentheses to make a function do what it is supposed to do).

```
#Asks the name and greets you

my_name = input("What is your name? ")
print("Hello,", my_name)
```

So, this short program will first ask you your name. You will need to type it in a box and then press Enter. After it you will see a greeting. You should try this code and see the result. What is new here? We already know what `print` and `input` do. The new part here is the variable `my_name` which first stores the string you type and after it is used in the function `print` to make a full sentence. Hence it is the time to discuss *variables* in Python (for those proficient with Python: I am well aware that in Python programming language there is, strictly speaking, no such thing as "variables," but this does not belong to this course).

1.2 Variables

In mathematics, we often write "Let $\boldsymbol{A}$ be a square matrix with real entries...", and after it, when we see the symbol $\boldsymbol{A}$, we understand what it means, without any additional information. In other words, we first *define* what letter $\boldsymbol{A}$ means, and after it do not bother to repeat the definition itself, seeing $\boldsymbol{A}$ is enough. We can similarly look at the variables in Python. We *create* (*define*) our variable (create a name) using the assignment operator `=`, such that the name should be on the left side of this operator and the definition on the right. Remember though, that since it is the computer we make a definition for, we should be a little more concrete with our definitions, e.g., writing `a = b` in the case when we did not define `b` previously, will return an error. Here are some self-explanatory examples. You should supplement them with your own ones.

(Here and below, when I type something like `>>>` it means that I am working in Jupyter notebook in an interactive regime, and the text after `>>>` is what I type and execute. The output appears

without symbols >>>.)

```
>>> a = 5
>>> a
5
>> b = 3
>> b
3
>> a + b
8
>> c = 10.0
>> a + c
18.0
>> my_name = "Artem Novozhilov"
>> my_name
'Artem Novozhilov'
```

One point here to remember is that when Python creates a variable (that is, puts together a name and an object that this name refers to), it also determines what type this variable has. You can see the type of a variable using function `type`. In the following I assume that variables `a,b,c` already defined as above.

```
>>> type(a)
int
>>> type(c)
float
>>> type(my_name)
str
```

Note that `type(1)` is `int` (integer) and `type(1.0)` is `float` (a rational number), that is, these are not the same internally, we will talk much more about it in a due course. `str` means type string. We in general care about types of our variables because certain types can be used in the same expression, and some cannot.

```
>>> a+c
18.0
>>> a+my_name
---------------------------------------------------------------------------
TypeError                                 Traceback (most recent call last)
<ipython-input-16-161b5ca6c2af> in <module>
----> 1 a+my_name

TypeError: unsupported operand type(s) for +: 'int' and 'str'
```

So I can naturally add integers and rational numbers (the result always will be a rational number), but an attempt to add an integer and a string results in an error. You should try to add two strings and see the result.

For the purposes of our course, in addition to `int`, `float`, and `str`, we will need types `bool` (boolean) and `list` (arrays) (I discuss them below). Note that there are other data types that I do

not discuss.

Naturally, for our numerical types we can use the usual arithmetic operations

```
>> a = 12
>> b = 5
>> a + b   #addition
17
>> a * b   #multiplication
60
>> a - b   #subtraction
7
>> a / b   #division
2.4
>> a // b  #floor division
2
>> -12 // 5 #a // b = q if a = b*d+r, where 0<=r<b
-3
>> a % b   #remainder
2
>> a ** b  #exponintiation
248832
```

Remember that you are pretty free to choose the names for your variables, they can contain letters, numbers, and underscore, and just cannot start with numbers. You should avoid, however, using the built-in function names as the names for your variables, and try to make the names of your variables descriptive, this will make reading your code easier.

Before we move on, let me implement a simple algorithm of switching the values of two given variables in three different ways.

```
#Algorithm 1 to switch the values of two variables

#initial values
a = 5
b = 2

print(f"Initial values are a = {a}, b = {b}.")

#I introduce a temporary variable tmp
tmp = a
a = b
b = tmp

print(f"The result is a = {a}, b = {b}.")

#Algorithm 2 to switch the values of two variables.
#Now I will not use a temporary variable

#initial values
a = 5
b = 2
```

```
print(f"Initial values are a = {a}, b = {b}.")

a = a + b #Make sure you go step by step
b = a - b #and understand what happens here.
a = a - b #I do not think it is that obvious.

print(f"The result is a = {a}, b = {b}.")

#Algorithm 3 to switch the values of two variables using Python syntax

#initial values
a = 5
b = 2

print(f"Initial values are a = {a}, b = {b}.")

a, b = b, a #Switching

print(f"The result is a = {a}, b = {b}.")

#Output is the same in all three cases
Initial values are a = 5, b = 2.
The result is a = 2, b = 5.
```

First note the syntax for `print` function. You put `f` in front of your string and then you can use inside curly brackets the names of the variables that have some values that you'd like to print. It is a very convenient syntax when you need to print a number of your variables inside one sentence.

Second, you should carefully analyze all three algorithms. Algorithm 1 is the most natural for my taste (mostly due to my little experience actually programming in Python) and should be most readable. We need to put the value of `b` into `a` and vice versa. For this goal I create a new temporary variable `tmp`, which first keeps the value of `a`. Now I can change `a` without loosing its value, and then assign its value to `b`. Simple as that.

In Algorithm 2 my gain is that I am not creating any new variables (I am saving the memory!), but the price is that very few people looking at these three lines will immediately say what is done in this part of the code.

In Algorithm 3 I use something, which may look a little strange if you did not use Python before, but is a very powerful way to make multiple assignments at the same time. The idea is that the number of variables on the left should be equal to the number of variables with some values on the right of the equality sign, Python will do the rest without any mistakes.

So, what is the punch line? First, there are *always* (quite) different ways to write program code. Do not try to come up with "best possible code," it probably does not exist. Second, prefer easy to understand algorithms to tricky ones (these are also sometimes very useful, for an extreme example go to youtube and search for "fast inverse square root"), especially if you do not have much experience in programming. And finally, as you learn a programming language, try to use the tools it provides, they are usually very well optimized and exist for a reason. Therefore, if you clearly understand what Algorithm 3 does, it is probably a good idea to use it in your code.

1.3 Loops and type `bool`

As I mentioned at the beginning, for numerical computations we are often required to perform a ridiculous number of similar computations. It would be unwise to devote a separate string in our program for each such computation and this is where a possibility to do something until a certain condition is met comes into play.

There are two ways in Python to perform computations in a loop: `while` loop and `for` loop. Let me start with the former one, since it is the most general loop structure in Python.

Assume that I need to print squares of first 10 natural numbers. Here is how I can do it.

```
n = 10     #how many natural numbers I will square
index = 1  #auxiliary variable which will range from 1 to n

while index <= n: #the body of the loop, note the indentation below

    square = index * index
    print(f"The square of {index} is {square}.")
    index = index + 1

#the output
The square of 1 is 1.
The square of 2 is 4.
The square of 3 is 9.
The square of 4 is 16.
The square of 5 is 25.
The square of 6 is 36.
The square of 7 is 49.
The square of 8 is 64.
The square of 9 is 81.
The square of 10 is 100.
```

I start with the keyword `while`, after which I must have a test condition which can be either `True` or `False`. If this condition is `False`, this stops any action in the loop whereas if this condition is `True` I follow the colon `:` and all the commands below `while` are executed. Note that there are four spaces to the right for the commands below `while`. This is done to see what actually belongs to this loop and is mandatory. If you do not make the correct indentation in Python, it will return an error.

Let us see several first steps. Initially variable `n` is 10 and variable `index` is set to 1. At the start of the loop Python checks whether `n<=index`, which is obviously `True`, and hence proceeds to the body of the loop. Inside the body another variable `square` is created, which is simply `index` squared. After it the result in shown on the screen using function `print`. Finally, the value of `index` is increased by 1 and the program goes back to the top to check whether `index <= n`, etc.

Sometimes by mistake we write a loop which never ends (the condition stays always true). Here is an example:

```
n = 0

while n > -1:
    n = n + 1
```

To interrupt such computation in a Jupyter notebook you should click the black square symbol.

A few words about tests (something that goes after `while` keyword). Such logical expressions in Python have type `bool` and can be either `True` or `False`. To generate it you can use the usual comparisons `<, >, <=, >=`, to check whether the right hand side is equal to the left hand side use `==`, for not equal use `!=`, and also logical operators `and`, `or`, and `not`, which act according to the standard rules that you all saw in, e.g., Math 270 (recall *truth tables*).

```
>>>type(2 < 4)
bool
>>>2 < 4
True
>>>2 > 4
False
>>> 2 == 4
False
>>> 2 != 4
True
>>> not True
False
>>> True or False
True
>>> True and False
False
```

Another commonly used loop structure in Python is `for`, which is also present in pretty much any other procedural programming language. It has, however, its own Pythonic peculiarities and allows to write a very clean and concise code. Jumping a little ahead, one of the most basic data structure in Python is the list, which is written as `my_list = [1, 2, 3, 4, 5, 6, 7, 8, 9, 10]`, i.e., data divided by commas (I will talk much more about lists in the next section). I can access individual data points in a list using index, which always starts with 0. So, `my_list[0]` is 1, `my_list[5]` is 6, and `my_list[9]` is 10 in my case. Using built-in function `len` I can find out how long my list is. In my case `len(my_list)` will return 10 (there are ten elements).

Now I will print first ten squared natural numbers using Python starting with a very bad (initially) code.

```
my_list = [1, 2, 3, 4, 5, 6, 7, 8, 9, 10]
indexes = [0, 1, 2, 3, 4, 5, 6, 7, 8, 9] #auxillary list of indexes

for index in indexes:
    square = my_list[index] * my_list[index]
    print(f"The square of {my_list[index]} is {square}.")
```

Note the syntax of `for` loop. Technically speaking, it works over *iterable objects*, a list is one example of such an object. Looking just a few seconds at the example above, you should ask yourself: Do I really need the list of indexes here? And the answer is "no." A cleaner version would be

```
my_list = [1, 2, 3, 4, 5, 6, 7, 8, 9, 10]

for index in my_list:
```

```
    square = index * index
    print(f"The square of {index} is {square}.")
```

note absence of any indexes in the code. One thing which still may be questionable if the fact that I typed my list manually. For numerical lists of this nature there is a very useful function that produces an arithmetic progression and has the form `range(start,stop,step)`, where start is the initial value, stop is the final non included(!) value, and step is naturally the step of this progression (so, mathematically we range over the interval $[start, stop)$). So, here is my final solution.

```
for index in range(1,11):
    square = index * index
    print(f"The square of {index} is {square}.")
```

When I do not specify `step` explicitly, it is assumed to be 1. Sometimes also abbreviation `range(5)` is useful, which gives you an arithmetic progression starting at 0 and ending at 4 (5 is not included) with step 1.

We can have nested loops, both `while` and `for`. For instance, the code

```
for i in range(5):
    for j in range(4):
        print(f"({i}, {j})")
```

will produce all pairs starting with (0,0), (0,1),... and ending with (4,2), (4,3).

1.4 Branching

Very often we need to decide which direction our program will go depending on some logical condition. For this purpose one usually uses `if-else` expression, here is an example, in which you are requested to type a number, and the program will decide whether the number even or odd.

```
a = int(input()) #put into the variable a the constant you type
                 #note that here I need to convert the result of your typing,
                 #which is a string, into an integer, I use function int()

if a % 2 == 0: #if the condition after if is True, the code says it is even
    print(f"{a} is an even number")
else:          #otherwise, we assume that the number is odd
    print(f"{a} is an odd number")
```

As with loops, we can have as many nested ifs as required. The code, however, quickly becomes difficult to read. For instant, assume that you are asked to type two coordinates of a point on a plane, and the program should tell you to which quadrant this point belongs to. One complication is that one (or both) coordinates can be zero, and in this case the program should indicate that the point is on one of the axes. An obvious but ugly solution may look like the following.

```
x_coord = int(input("x="))
y_coord = int(input("y="))

if x_coord > 0:
```

```
    if y_coord > 0:
        print("First quadrant")
    else:
        if y_coord <0:
            print("Fourth quadrant")
        else:
            print("On the axes")
else:
    if x_coord < 0:
        if y_coord > 0:
            print("Second quadrant")
        else:
            if y_coord <0:
                print("Third quadrant")
            else:
                print("On the axes")
    else:
        print("On the axes")
```

A cleaner Python solution would use the keyword `elif`, which is an abbreviation from `else` and `if`, which allows one to check several possible alternative in a readable and simple way. (Here parenthesis around logical conditions are not necessary, but for some may simplify reading of the code.)

```
x_coord = int(input("x="))
y_coord = int(input("y="))

if (x_coord > 0) and (y_coord > 0):
    print("First quadrant")

elif (x_coord > 0) and (y_coord < 0):
    print("Fourth quadrant")

elif (x_coord < 0) and (y_coord < 0):
    print("Third quadrant")

elif (x_coord < 0) and (y_coord > 0):
    print("Second quadrant")

else:
    print("On the axes")
```

1.5 Quick review

We did cover quite a few points in this introductory review of Python syntax. Let me summarize them.

We discussed

1. built-in functions `print`, `input`, `type`, `len`;
2. variables and the rules to name variables;

3. types `int`, `float`, `str`, `bool`, `list`;
4. multiple assignment `a, b = c, d`;
5. arithmetic operations `+, -, *, /, //, %, **`;
6. constants `True` and `False` of type `bool`;
7. logical operators `<, <=, >, >=, ==, !=, and, or, not`;
8. loops of the form `while`;
9. loops of the form `for`, which work great for iterable objects;
10. a built-in function to produce an arithmetic progression `range(s,s,s)`;
11. explicit type conversion `int("123")`;
12. branchings of the form `if-else` and `if-elif-else`;
13. the rule to have four spaces to identify the code block that belongs to, e.g., a loop or to an `if` statement.

If any of the command is unclear, please go back to its description and experiment in your Jupyter notebook. We will see more in the next section and finally you will have a chance to work on a number of programming problems.

I also recommend to those who feel a little intimidated by the amount of new material in this section to work through Appendix A in the course textbook (up to Section A.5) and try all the examples and exercises there.